A New Proof of P-time Completeness of Linear Lambda Calculus

National Institute of Advanced Industrial Science and Technology (AIST),
1-1-1 Umezono, Tsukuba, Ibaraki, 305-8563 Japan

Satoshi Matsuoka

January 15, 2013

Abstract

We give a new proof of P-time completeness of Linear Lambda Calculus, which was originally given by H. Mairson in 2003. Our proof uses an essentially different Boolean type from the type Mairson used.
Moreover the correctness of our proof can be machined-checked using an implementation of Standard ML.

1 Introduction

In [Mai04], H. Mairson gave a proof of P-time completeness of Linear Lambda Calculus. It is an excellent exercise of linear functional programming. The crucial point of the proof is that the copy function of truth values is representable by a linear term: this is relatively easy in Affine Lambda Calculus as shown in [Mai04], but quite difficult in Linear Lambda Calculus. So, the key issue there is to avoid the use of the weakening rule. The issue was also treated from a different angle in [Mat07], which established typed Böhm theorem without the weakening rule.
In this paper we give a new proof of P-time completeness of Linear Lambda Calculus. Our proof is different from that of [Mai04] in the following points:

- In [Mai04] Mairson used the base Boolean type $\mathbb{B}_{\mathrm{MH}} = p \multimap p \multimap (p \multimap p \multimap p) \multimap p$ while we use $\mathbb{B} = (p \multimap p) \multimap (p \multimap p) \otimes (p \multimap p)$. Although both have two normal forms, they are different because while $\mathbb{B}_{\mathrm{MH}}$ reduces to itself by the *linear distributive transformation* given in Section 3 of [Mat07] (which was called third order reduction in [Mat07]), $\mathbb{B}$ reduces to

$$\mathbb{B}_{\mathrm{red}} = p \multimap p \multimap (p \multimap p) \multimap (p \multimap p \multimap p) \multimap p,$$

  which has six normal forms.

- All the two variable functions that can be representable over $\mathbb{B}_{\mathrm{MH}}$ *without any polymorphism* are only *exclusive or* and its negation, but in $\mathbb{B}$ they are all the boolean functions except for exclusive or and its negation, i.e., fourteen functions.

- our proof is also an interesting application of the linear distributive transformation.

As in [Mai04], our proof is also machine-checkable: all the linear λ-terms in this paper are also well-formed expressions of Standard ML [MTHM97]. So the reader may confirm the correctness of our proof using an implementation of Standard ML. We used the interactive system of Standard ML of New Jersey.

2 Typing Rules

We give our term assignment system for Linear Lambda Calculus. Our system is based on Natural Deduction, e.g., given in [Tro92], which is equivalent to the system based on Sequent Calculus or proof nets in [Gir87] (see [Tro92]). Its notation is unusual in the Linear Logic community, but its purpose is to make our proof machine-checkable.

Types

$$\texttt{A} ::= \texttt{'a} \mid \texttt{A1*A2} \mid \texttt{A1->A2}$$

The symbol `'a` stands for a type variable. On the other hand `A1*A2` stands for the tensor product $\texttt{A1} \otimes \texttt{A2}$ and `A1->A2` for the linear implication $\texttt{A1} \multimap \texttt{A2}$.

Terms We use `x,y,z` for term variables, $\vec{\texttt{x}}, \vec{\texttt{y}}, \vec{\texttt{z}}$ for finite lists of term variables, and `t,s` for general terms.

Term Assignment System

$$\frac{}{\texttt{x:A} \vdash \texttt{x:A}}$$

$$\frac{\texttt{x:A}, \vec{\texttt{y}}\texttt{:}\Gamma \vdash \texttt{t:B}}{\vec{\texttt{y}}\texttt{:}\Gamma \vdash \texttt{fn x=>t:A->B}} \qquad \frac{\vec{\texttt{x}}\texttt{:}\Gamma \vdash \texttt{t:A->B} \quad \vec{\texttt{y}}\texttt{:}\Delta \vdash \texttt{s:A}}{\vec{\texttt{x}}\texttt{:}\Gamma, \vec{\texttt{y}}\texttt{:}\Delta \vdash \texttt{ts:B}}$$

$$\frac{\vec{\texttt{x}}\texttt{:}\Gamma \vdash \texttt{s:A} \quad \vec{\texttt{y}}\texttt{:}\Delta \vdash \texttt{s:B}}{\vec{\texttt{x}}\texttt{:}\Gamma, \vec{\texttt{y}}\texttt{:}\Delta \vdash \texttt{(s,t):A*B}} \qquad \frac{\vec{\texttt{x}}\texttt{:}\Gamma \vdash \texttt{s:A*B} \quad \texttt{x:A},\texttt{y:B}, \vec{\texttt{y}}\texttt{:}\Delta \vdash \texttt{t:C}}{\vec{\texttt{x}}\texttt{:}\Gamma, \vec{\texttt{y}}\texttt{:}\Delta \vdash \texttt{let val (x,y)=s in t end:C}}$$

Moreover, function declaration
`fun f x1 x2 ··· xn = t`
is interpreted as the following term:
`f = fn x1 => (fn x2 => ( ··· (fn xn => t) ··· ))`
We only consider closed term (or combinator)
$\vdash$ `t:A`.
The following proposition is proved easily by structural induction:

Proposition 2.1 *If* `x1:A1,...,xn:An|-t:B` *then*
`fun f x1 x2 ··· xn = t`
is a well formed function declaration of Standard ML.

Term Reduction Rules Two of our reduction rules are
(β): `(fn x=>t)s` $\Rightarrow$ `t[s/x]`
($\otimes$-red): `let val (x,y)=(u,v) in w end` $\Rightarrow$ `w[u/x,v/y]`
In fact, in Standard ML, `s,u,v` must be *values* in order for these rules to be applied. But Linear Lambda Calculus satisfies SN and CR properties. So we don't need to care the evaluation order. Then note that if a function `f` is defined by
`fun f x1 x2` $\cdots$ `xn = t`
and
`x1:A1,...,xn:An|-s:B`, `|-t1:A1`, ..., `|-tn:An`
then, we have
`f t1` $\cdots$ `tn` $\Rightarrow$ `t[t1/x1,...,tn/xn]`.
Moreover we need the following reduction for a theoretical reason, which is absent from Standard ML:
(η): `t` $\Rightarrow$ `(fn x => t x)`
In the following $=_{\beta\eta}$ denotes the congruence relation generated by the three reduction rules.

3 Review of Mairson's Proof

In this section we review the proof in [Mai04] briefly. Below by normal forms we mean $\beta\eta$-long normal forms. The basic construct is the following term:

```
- fun Pair x y z = z x y;
val Pair = fn :  'a -> 'b -> ('a -> 'b -> 'c) -> 'c
```

Using this, we define `True` and `False`:

```
- fun True x = Pair x y;
val True = fn :  'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- fun False x = Pair y x;
val True = fn :  'a -> 'b -> ('b -> 'a -> 'c) -> 'c
```

Note that these are the normal forms of $\mathbb{B}_{\mathrm{MH}}$. In order to define the term `Copy` two auxiliary terms are needed:

```
- fun I x = x;
val I = fn :  'a -> 'a
- fun id B = B I I I ;
val id = fn :  (('a -> 'a) -> ('b -> 'b) -> ('c -> 'c) -> 'd) -> 'd
```

The formal argument `B` is supposed to receive `True` or `False`. It is easy to see that

$$\texttt{id True} \Rightarrow^{*} \texttt{I}, \qquad \texttt{id False} \Rightarrow^{*} \texttt{I},$$

Then the term `Copy` is defined as follows:

```
- fun Copy P = P (Pair True True) (Pair False False)
(fn U => fn V =>
U (fn u1 => fn u2 =>
V (fn v1 => fn v2 =>
((id v1) u1, (id v2) u2)))))
```

We omit its type since it is too long. The formal argument `P` is supposed to receive `True` or `False`. While [Mai04] uses continuation passing style, the above term not since we have the $\otimes (= *)$-connective and can do a direct encoding using this connective. Then

```
Copy True;
val it=(fn,fn):('a -> 'b -> ('a -> 'b -> 'c) -> 'c)*('d -> 'e -> ('d -> 'e -> 'f) -> 'f)
Copy False;
val it=(fn,fn):('a -> 'b -> ('b -> 'a -> 'c) -> 'c)*('d -> 'e -> ('e -> 'd -> 'f) -> 'f)
```

(in fact, since SML/NJ does not allow any function values, it gives warnings, but the results are basically the same). These are `(True, True)` and `(False, False)` respectively since

$$\begin{aligned} \texttt{Copy True} &\Rightarrow^* ((\texttt{id False})\ \texttt{True},\ (\texttt{id False})\ \texttt{True}) \\ \texttt{Copy False} &\Rightarrow^* ((\texttt{id True})\ \texttt{False},\ (\texttt{id True})\ \texttt{False}) \end{aligned}$$

The basic observation here is that

- the type of `P` is unifiable with that of both `True` and `False`;
- the types of `Copy True` and `Copy False` are desirable ones, i.e., both have $\mathbb{B}_{\mathrm{MH}} \otimes \mathbb{B}_{\mathrm{MH}}$ as a instance.

Since the *and* gate can be defined similarly and more easily and the *not* gate without any ML-polymorphism, it is concluded that all the boolean gates can be defined over $\mathbb{B}_{\mathrm{MH}}$.

4 A Partial Solution

In this section we present our failed attempt.
Let the following two terms be `True'` and `False'`:

```
fun True' x y f = Pair x (f y);
val True' = fn :  'a -> 'b -> ('b -> 'c) -> ('a -> 'c -> 'd) -> 'd
fun False' x y f = Pair (f x) y;
val False' = fn :  'a -> 'b -> ('a -> 'c) -> ('c -> 'b -> 'd) -> 'd
```

Both are normal forms of $\mathbb{B}_{\mathrm{red}} = p \multimap p \multimap (p \multimap p) \multimap (p \multimap p \multimap p) \multimap p$, which have the six normal forms. Below we define a copy function for them as in the previous section. In order to do that, we need several auxiliary terms:

```
fun not' f x y g h = f y x g (fn u => fn v => (h v u));
val not' = fn
:  ('a -> 'b -> 'c -> ('d -> 'e -> 'f) -> 'g)
-> 'b -> 'a -> 'c -> ('e -> 'd -> 'f) -> 'g
```

The term `not'` is the *not* gate for the new boolean values.

```
fun swap f g = f (fn u => fn v => g v u);
val swap = fn :  (('a -> 'b -> 'c) -> 'd) -> ('b -> 'a -> 'c) -> 'd
```

We note that

$$\begin{aligned} \texttt{swap}\,(\texttt{Pair False}'\ \texttt{True}')\ \texttt{g} &=_{\beta\eta} (\texttt{Pair False}'\ \texttt{True}')(\texttt{fn u => fn v => g u v}) \\ &=_{\beta\eta} \texttt{g True}'\ \texttt{False}' =_{\beta\eta} \texttt{Pair True}'\ \texttt{False}'\ \texttt{g} \end{aligned}$$

The term `newid` is similar to `id`, but receives four arguments:

```
fun newid B' = B' I I I I;
```

```
val newid = fn
:  (('a -> 'a) -> ('b -> 'b) -> ('c -> 'c) -> ('d -> 'd) -> 'e) -> 'e
```

The term `constNot` is also similar to `id`, but always returns `not'`:

```
fun constNot B' = B' I not' I I;
val constNot = fn
:  (('a -> 'a)
-> (('b -> 'c -> 'd -> ('e -> 'f -> 'g) -> 'h)
-> 'c -> 'b -> 'd -> ('f -> 'e -> 'g) -> 'h)
-> ('i -> 'i) -> ('j -> 'j) -> 'k)
-> 'k
```

The formal argument `B'` in `newid` and `constNot` is supposed to receive `True'` and `False'`. We can easily see

$$\begin{array}{cc} \texttt{newid True}' \Rightarrow^* \texttt{I}, & \texttt{newid False}' \Rightarrow^* \texttt{I}, \\ \texttt{constNot True}' \Rightarrow^* \texttt{not}, & \texttt{constNot False}' \Rightarrow^* \texttt{not} \end{array}$$

Under the preparation above, we can define `Copy'` as follows:

```
fun Copy' P' = P' (Pair False' True') (Pair False' True') swap
(fn U => fn V =>
U (fn u1 => fn u2 =>
V (fn v1 => fn v2 =>
((constNot v1) u1, (newid v2) u2))));
```

Again we omit the type. The formal parameter `P` is supposed to receive `True'` or `False'`. Then

$$\begin{array}{rcl} \texttt{Copy}' \ \texttt{True}' & \Rightarrow^* & ((\texttt{constNot True}) \ \texttt{False}, (\texttt{newid False}) \ \texttt{True}) \\ & \Rightarrow^* & (\texttt{True}, \texttt{True}) \\ \texttt{Copy}' \ \texttt{False}' & \Rightarrow^* & ((\texttt{constNot False}) \ \texttt{True}, (\texttt{newid True}) \ \texttt{False}) \\ & \Rightarrow^* & (\texttt{False}, \texttt{False}) \end{array}$$

Unfortunately we could not find a term that represents the *and* gate over `True'` and `False'`. So, we must find a similar, but different substitute. Fortunately we have found a solution described in the next section.

5 Our Solution

Our solution uses the type $\mathbb{B} = (p \multimap p) \multimap (p \multimap p) \otimes (p \multimap p)$, which has the two normal forms:

```
fun True" x = (fn z => z, fn y => x y);
val True" = fn :  ('a -> 'b) -> ('c -> 'c) * ('a -> 'b)
fun False" x = (fn y => x y, fn z => z);
val False" = fn :  ('a -> 'b) -> ('a -> 'b) * ('c -> 'c)
```

The linear distributive transformation (which was called third-order reduction in [Mat07]) turn $\mathbb{B}$ into $\mathbb{B}_{\text{red}}$. The next term is its internalized version:

```
fun LDTr h x y f z
= let val (k, l) = h f in z (k x) (l y) end;
```

```
val LDTr = fn
:  ('a -> ('b -> 'c) * ('d -> 'e))
-> 'b -> 'd -> 'a -> ('c -> 'e -> 'f) -> 'f
```

Then

$$\texttt{LDTr True}'' =_{\beta\eta} \texttt{True}', \qquad \texttt{LDTr False}'' =_{\beta\eta} \texttt{False}'$$

Our *not* gate for `True″` and `False″` is

```
fun not″ h f = let val (k, l) = h f in (l, k) end;
val not″ = fn :  ('a -> 'b * 'c) -> 'a -> 'c * 'b
```

Then

$$\texttt{not}''\ \texttt{True}'' =_{\beta\eta} \texttt{False}'', \qquad \texttt{not}''\ \texttt{False}'' =_{\beta\eta} \texttt{True}''$$

Moreover we can write down a *and* gate for them as follows:

```
fun and″ f g h = let val (u, v) = g (fn k => h k) in
(let val (x, y) = f (fn w => v w) in
(fn s => x (u s), fn t => y t) end) end;
val and″ = fn
:  (('a -> 'b) -> ('c -> 'd) * ('e -> 'f))
-> (('g -> 'h) -> ('i -> 'c) * ('a -> 'b))
-> ('g -> 'h) -> ('i -> 'd) * ('e -> 'f)
```

Note that the definition of `and″` does not use any ML-polymorphism. Then

$$\begin{array}{cc} \texttt{and}''\ \texttt{True}''\ \texttt{True}'' =_{\beta\eta} \texttt{True}'', & \texttt{and}''\ \texttt{False}''\ \texttt{False}'' =_{\beta\eta} \texttt{False}'', \\ \texttt{and}''\ \texttt{True}''\ \texttt{False}'' =_{\beta\eta} \texttt{False}'', & \texttt{and}''\ \texttt{False}''\ \texttt{True}'' =_{\beta\eta} \texttt{False}'' \end{array}$$

Next, we define a copy function for `True″` and `False″`. In order to do that, we need a modified version of `constNot`:

```
fun constNot″ B″ = B″ I not″ I I;
```

Then we can easily see

$$\begin{array}{cc} \texttt{newid True}'' \Rightarrow^* \texttt{I}, & \texttt{newid False}'' \Rightarrow^* \texttt{I}, \\ \texttt{constNot}''\ \texttt{True}'' \Rightarrow^* \texttt{not}, & \texttt{constNot}''\ \texttt{False}'' \Rightarrow^* \texttt{not} \end{array}$$

Under the preparation above, we can define `Copy″`, which is a modified version of `Copy′` as follows:

```
fun Copy″ P
= LDTr P (Pair False″ True″) (Pair False″ True″) swap
(fn U => fn V =>
U (fn u1 => fn u2 =>
V (fn v1 => fn v2 =>
((constNot″ (LDTr v1)) u1, (newid (LDTr v2)) u2))));
```

Then

$$\texttt{Copy}''\ \texttt{True}'' \Rightarrow^* (\texttt{True}'', \texttt{True}''), \qquad \texttt{Copy}''\ \texttt{False}'' \Rightarrow^* (\texttt{False}'', \texttt{False}'')$$

From what precedes we can conclude that we can represent all the boolean gates over $\mathbb{B}$.

6 Concluding Remarks

In this paper we showed that $\mathbb{B}_{\mathrm{MH}}$ is not the only choice in order to establish P-time completeness of Linear Lambda Calculus. We note that we found the term `and″` manually using proof nets syntax (and then translating the proof net into `and″`), but `Copy′` and `Copy″` interactively with Standard ML of New Jersey.
From our result a natural question comes up: which linear type other than $\mathbb{B}_{\mathrm{MH}}$ and $\mathbb{B}$ and its two normal forms establishes P-time completeness of Linear Lambda Calculus? For example it is unlikely that $\mathbb{B}' = p \multimap (p \multimap p) \multimap (p \multimap p) \multimap p$ and its two normal forms establish that. But it is an easy exercise to show that $(p \otimes p) \multimap (p \otimes p)$ and its two normal forms can do that.
As wrote before, we could not prove that $\mathbb{B}_{\mathrm{red}}$ and its normal forms `True′` and `False′` establish P-time completeness of Linear Lambda Calculus. But we also could not prove that they cannot establish that. At this moment we do not have any idea to do that. Our type $\mathbb{B}$ and its generalization $(p \multimap p) \multimap \overbrace{(p \multimap p) \otimes \cdots \otimes (p \multimap p)}^{n}$ have further interesting properties. For example we can establish weak typed Böhm theorem over $\mathbb{B}$. But the subject is beyond the scope of this paper, and will be discussed elsewhere.

References

[Gir87] J.-Y. Girard. Linear Logic, Theoretical Computer Science, 50, 1-102, 1987.

[Mat07] Satoshi Matsuoka. Weak Typed Böhm Theorem on IMLL, Annals of Pure and Applied Logic, 145(1): 37-90, 2007.

[Mai04] H.G. Mairson. Linear Lambda Calculus and PTIME-completeness, Journal of Functional Programing, 14(6): 623-633, 2004.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The Definition of Standard ML (Revised). MIT Press, 1997.

[Tro92] A.S. Troelstra. Lectures on Linear Logic, CSLI, 1992.